Lecture 5

Recap: #SAT Problem

- Let φ be a Boolean formula of size S over n variables.
- Goal: Compute $\sum_{x \in \{0,1\}^n} \varphi(x)$.

Recap: #SAT Problem

- Let φ be a Boolean formula of size S over n variables.
- Goal: Compute $\sum_{x \in \{0,1\}^n} \varphi(x)$.

- Protocol:
- Let g be an extension polynomial of φ.
- Apply the sum-check protocol to compute $\sum_{x \in \{0,1\}^n} g(x)$.

Recap: #SAT Problem

- Let φ be a Boolean formula of size S over n variables.
- Goal: Compute $\sum_{x \in \{0,1\}^n} \varphi(x)$.

- Protocol:
- Let g be an extension polynomial of φ.
- Apply the sum-check protocol to compute $\sum_{x \in \{0,1\}^n} g(x)$.
  - Note: in final round of sum-check, V needs to compute $g(r)$ for some randomly chosen r in $\boldsymbol{F^n}$.
    - To control V's runtime, we need this to be fast.

Recap: #SAT Problem

- Let φ be a Boolean formula of size S over n variables.
- Goal: Compute $\sum_{x \in \{0,1\}^n} \varphi(x)$.

- Protocol:
- Let g be an extension polynomial of φ.
- Apply the sum-check protocol to compute $\sum_{x \in \{0,1\}^n} g(x)$.
  - Note: in final round of sum-check, V needs to compute $g(r)$ for some randomly chosen r in $\boldsymbol{F^n}$.
    - To control V's runtime, we need this to be fast.
  - To control communication and P and V's runtime, we need g to be "low-degree".

Recap: #SAT Problem

- Let φ be a Boolean formula of size S over n variables.
- Goal: Compute $\sum_{x \in \{0,1\}^n} \varphi(x)$.

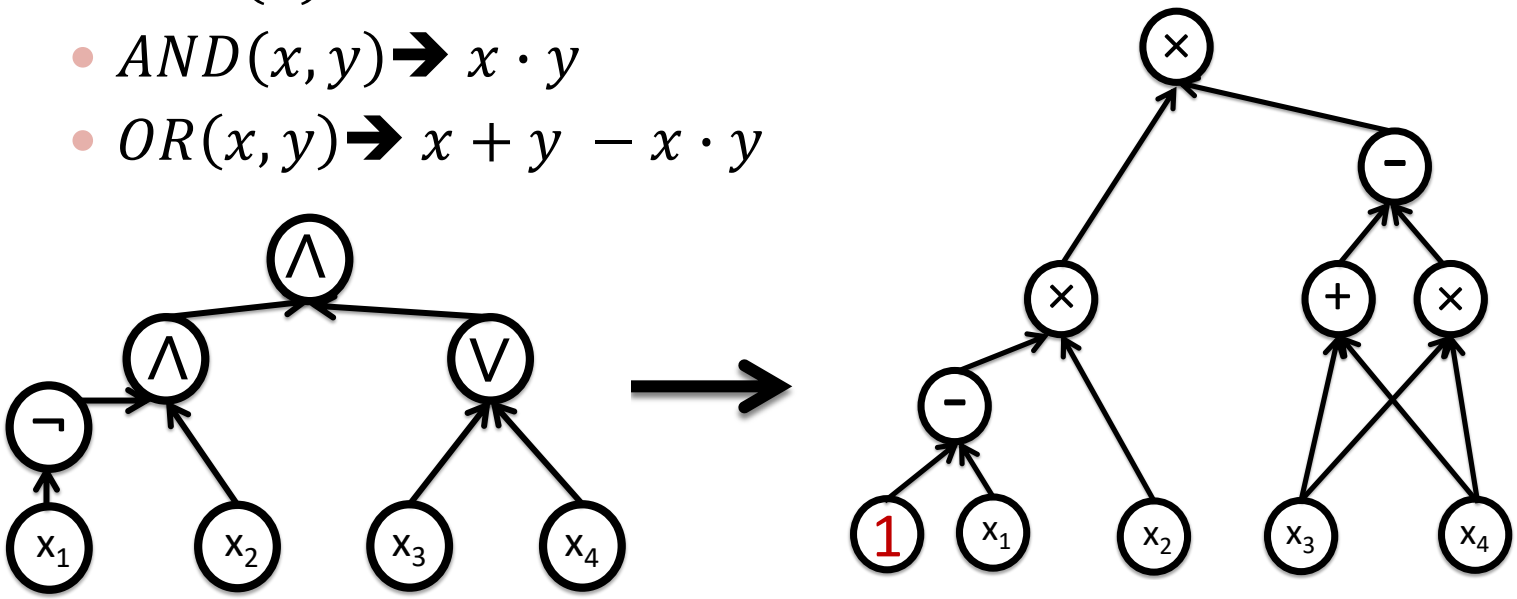
- Protocol:
- Let g be an extension polynomial of φ.
- Apply the sum-check protocol to compute $\sum_{x \in \{0,1\}^n} g(x)$.
  - Note: in final round of sum-check, V needs to compute $g(r)$ for some randomly chosen r in $\boldsymbol{F^n}$.
    - To control V's runtime, we need this to be fast.
  - To control communication and P and V's runtime, we need g to be "low-degree".
  - Key question: how to construct the extension polynomial g?

Arithmetization

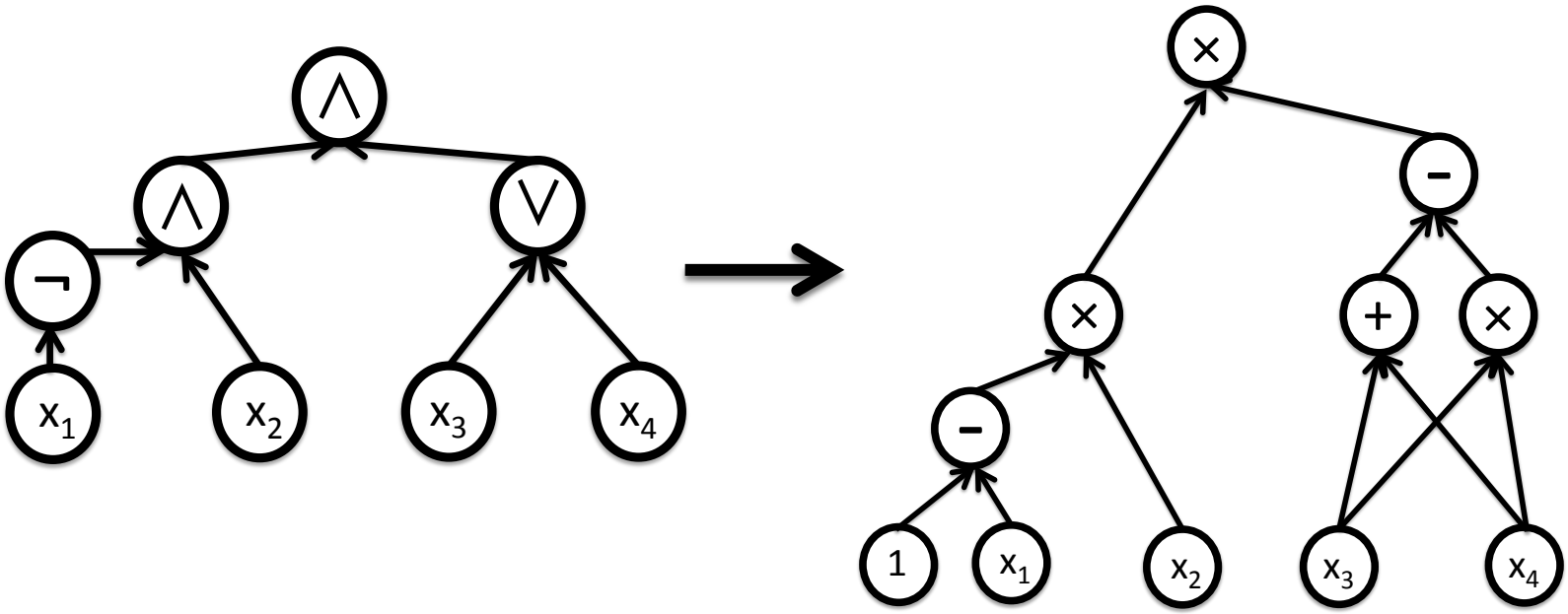
- Key question: how to construct the extension polynomial g?
- Answer: Arithmetize φ
  - i.e., replace φ with an **arithmetic** circuit computing extension g
    - Go gate-by-gate through φ, replacing each gate with the gate's multilinear extension.
    - $NOT(x) \rightarrow 1 - x$
    - $AND(x, y) \rightarrow x \cdot y$
    - $OR(x, y) \rightarrow x + y - x \cdot y$

Arithmetization

- Key question: how to construct the extension polynomial g?
- Answer: Arithmetize φ
  - i.e., replace φ with an **arithmetic** circuit computing extension g
    - Go gate-by-gate through φ, replacing each gate with the gate's multilinear extension.
    - $NOT(x) \rightarrow 1 - x$
    - $AND(x, y) \rightarrow x \cdot y$
    - $OR(x, y) \rightarrow x + y - x \cdot y$



Summary of Arithmetization



Transforming a Boolean formula φ of size S into an arithmetic circuit computing an extension g of φ.

Note: $\deg(g) \leq S$, and g can be evaluated at any input, gate by gate, in time $O(S)$.

Costs of #SAT Protocol Applied to g

- Let φ be a Boolean formula of size S over n variables, g the extension obtained by arithmetizing φ.

Rounds	Communication	V Time	P Time
n	P sends a degree S polynomial in reach round, V sends one field element in each round $\Rightarrow$ $O(S \cdot n)$ field elements sent in total.	• $O(S)$ time to process each of the n messages of P • $O(S)$ time to evaluate $g(r)$ $\Rightarrow$ $O(S \cdot n)$ time total	P evaluates g at $O(S \cdot 2^n)$ points to determine each message $\Rightarrow$ $O(S^2 \cdot n \cdot 2^n)$ time in total.

Details of Prover's Computation

- **Round i=1,2,…,** n : P sends **univariate** polynomial $s_i(X_i)$ claimed to equal:

$$H_i(X_i) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, X_i, b_{i+1,} b_{i+2,} \ldots, b_n)$$

- **Round i=1,2,…,** n : P sends **univariate** polynomial $s_i(X_i)$ claimed to equal:

$$H_i(X_i) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, X_i, b_{i+1}, b_{i+2}, \ldots, b_n)$$

- H_i has degree $d = O(S)$ because g has degree $O(S)$ in each of its n variables.
- To specify any degree d polynomial it suffices to specify the polynomial's evaluations at inputs in $[d] := \{0, 1, 2, \ldots, d\}$.

- **Round i=1,2,…,** n : P sends **univariate** polynomial $s_i(X_i)$ claimed to equal:

$$H_i(X_i) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, X_i, b_{i+1,} b_{i+2,} \ldots, b_n)$$

- H_i has degree $d = O(S)$ because g has degree $O(S)$ in each of its n variables.
- To specify any degree d polynomial it suffices to specify the polynomial's evaluations at inputs in $[d] := \{0, 1, 2, \ldots, d\}$.
- Note that for each $j \in [d]$,

$$H_i(j) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, j, b_{i+1,} b_{i+2,} \ldots, b_n)$$

is a sum of 2^{n-i} evaluations of g.

- **Round i=1,2,…,** n : P sends **univariate** polynomial $s_i(X_i)$ claimed to equal:

$$H_i(X_i) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, X_i, b_{i+1,} b_{i+2,} \ldots, b_n)$$

- H_i has degree $d = O(S)$ because g has degree $O(S)$ in each of its n variables.
- To specify any degree d polynomial it suffices to specify the polynomial's evaluations at inputs in $[d] := \{0, 1, 2, \ldots, d\}$.
- Note that for each $j \in [d]$,

$$H_i(j) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, j, b_{i+1,} b_{i+2,} \ldots, b_n)$$

is a sum of 2^{n-i} evaluations of g.

- Recalling that g can be evaluated at any input in $O(S)$ time, this means H_i can be evaluated at all inputs in $[d]$ in time $O(S^2 2^{n-i})$ time.

- **Round i=1,2,…,** n : P sends **univariate** polynomial $s_i(X_i)$ claimed to equal:

$$H_i(X_i) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, X_i, b_{i+1,} b_{i+2,} \ldots, b_n)$$

- H_i has degree $d = O(S)$ because g has degree $O(S)$ in each of its n variables.
- To specify any degree d polynomial it suffices to specify the polynomial's evaluations at inputs in $[d] := \{0, 1, 2, \ldots, d\}$.
- Note that for each $j \in [d]$,

$$H_i(j) := \sum_{b_i \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(r_1, r_2, \ldots, r_{i-1}, j, b_{i+1,} b_{i+2,} \ldots, b_n)$$

is a sum of 2^{n-i} evaluations of g.

- Recalling that g can be evaluated at any input in $O(S)$ time, this means H_i can be evaluated at all inputs in $[d]$ in time $O(S^2 2^{n-i})$ time.
- Across all n rounds, this is $O\left(\sum_{i=1}^{n} S^2 2^{n-i}\right) = O\left(S^2 2^n\right)$ time.

IP=PSPACE

- #SAT is a **#P**-complete problem.
  - Hence, the protocol we just saw implies that **every** problem in **#P** has an interactive proof with a polynomial time verifier.
- It is not much harder to show that this in fact holds for every problem in **PSPACE** [LFKN, Shamir].

IP=PSPACE

- #SAT is a **#P**-complete problem.
  - Hence, the protocol we just saw implies that **every** problem in **#P** has an interactive proof with a polynomial time verifier.
- It is not much harder to show that this in fact holds for every problem in **PSPACE** [LFKN, Shamir].
- But is this a **practical** result?

IP=PSPACE

- #SAT is a **#P**-complete problem.
  - Hence, the protocol we just saw implies that **every** problem in **#P** has an interactive proof with a polynomial time verifier.
- It is not much harder to show that this in fact holds for every problem in **PSPACE** [LFKN, Shamir].
- But is this a **practical** result?
  - No. The main reason: P's runtime.

IP=PSPACE

- #SAT is a **#P**-complete problem.
  - Hence, the protocol we just saw implies that **every** problem in **#P** has an interactive proof with a polynomial time verifier.
- It is not much harder to show that this in fact holds for every problem in **PSPACE** [LFKN, Shamir].
- But is this a **practical** result?
  - No. The main reason: P's runtime.
  - When applying the protocols of [LFKN, Shamir] even to very simple problems, the honest prover would require **superpolynomial** time.

IP=PSPACE

- #SAT is a **#P**-complete problem.
  - Hence, the protocol we just saw implies that **every** problem in **#P** has an interactive proof with a polynomial time verifier.
- It is not much harder to show that this in fact holds for every problem in **PSPACE** [LFKN, Shamir].
- But is this a **practical** result?
  - No. The main reason: P's runtime.
  - When applying the protocols of [LFKN, Shamir] even to very simple problems, the honest prover would require **superpolynomial** time.
  - The #SAT prover took time at least 2^n.
    - This seems unavoidable for #SAT, since we don't know how to even solve the problem in less than 2^n time.
    - But we can hope to solve "easier" problems without turning those problems into #SAT instances.

Doubly-Efficient Interactive Proofs

Doubly-Efficient Interactive Proof

- A doubly-efficient interactive proof for a problem is one where:
  - V runs in time linear in the input size.
  - P runs in polynomial time.

A Second Application of the Sum-Check Protocol

A Doubly-Efficient Interactive Proof for Counting Triangles

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ik}$.
- Fastest known algorithm runs in matrix-multiplication time, currently about $n^{2.37}$.

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ik}$.
- The Protocol:
  - View A as a function mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\boldsymbol{F}$.
  - Recall that $\tilde{A}$ denotes the multilinear extension of A.
  - Define the polynomial $g(X,Y,Z) = \tilde{A}(X,Y)\ \tilde{A}(Y,\mathrm{Z})\ \tilde{A}(X,\mathrm{Z})$
  - Apply the sum-check protocol to g to compute:

$$\sum_{(a,b,c) \in \{0,1\}^{3\log n}} g(a,b,c)$$

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ik}$.
- The Protocol:
  - View A as a function mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\boldsymbol{F}$.
  - Recall that $\tilde{A}$ denotes the multilinear extension of A.
  - Define the polynomial $g(X,Y,Z) = \tilde{A}(X,Y)\, \tilde{A}(Y,\mathrm{Z})\, \tilde{A}(X,\mathrm{Z})$
  - Apply the sum-check protocol to g to compute:

$$\sum_{(a,b,c) \in \{0,1\}^{3\log n}} g(a,b,c)$$

- Costs:
  - Total communication is $O(\log n)$, V runtime is $O(n^2)$, P runtime is $O(n^3)$.
  - V's runtime dominated by evaluating:

$$g(r_1,r_2,r_3) = \tilde{A}(r_1,r_2)\, \tilde{A}(r_2,r_3)\, \tilde{A}(r_1,r_3).$$

Third Application of the Sum-Check Protocol

A Doubly-Efficient Interactive Proof for Matrix Multiplication

[Thaler13]: Optimal IP For n x n MatMult

- Goal: Given $n \times n$ input matrices A, B over field $\boldsymbol{F}$, compute $C = A \cdot B$.

[Thaler13]: Optimal IP For n x n MatMult

- Goal: Given $n \times n$ input matrices A, B over field $\boldsymbol{F}$, compute $C = A \cdot B$.
- P simply determines the "right answer", and then P does $O(n^2)$ extra work to prove its correctness.
- Optimal runtime up to leading constant assuming no $O(n^2)$ time algorithm for MatMult.
- V runs in linear time (which is also optimal).

[Thaler13]: Optimal IP For n x n MatMult

- Goal: Given $n \times n$ input matrices A, B over field $\boldsymbol{F}$, compute $C = A \cdot B$.
- P simply determines the "right answer", and then P does $O(n^2)$ extra work to prove its correctness.
- Optimal runtime up to leading constant assuming no $O(n^2)$ time algorithm for MatMult.
- V runs in linear time (which is also optimal).

Problem Size	Naïve MatMult Time	Additional P time	V Time	Rounds	Protocol Comm
1024 x 1024	2.17 s	0.03 s	0.09 s	11	264 bytes
2048 x 2048	18.23 s	0.13 s	0.30 s	12	288 bytes

Comparison to Freivalds' Algorithm

- Recall that Freivalds in 1979 gave the following protocol for MatMult. To check $A \cdot B = D$:
  - V picks random vector x.
  - Accepts if $A \cdot (Bx) = Dx$.
  - **No** extra work for P, $O(n^2)$ time for V.

Comparison to Freivalds' Algorithm

- Recall that Freivalds in 1979 gave the following protocol for MatMult. To check $A \cdot B = D$:
  - V picks random vector x.
  - Accepts if $A \cdot (Bx) = Dx$.
  - **No** extra work for P, $O(n^2)$ time for V.
- Our big win: verifying algorithms that invoke MatMult, but aren't really interested in matrices.
  - E.g., Best-known subgraph-counting algorithms square the adjacency matrix, but are only interested in a single number.
  - Total communication for us is $O(\log n)$, Freivalds' is $\Omega(n^2)$.

MatMult Protocol: Technical Details

Notation

- Given $n \times n$ input matrices A, B over field $\boldsymbol{F}$, interpret A and B as functions mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\boldsymbol{F}$ via $A\left(i_1, \ldots, i_{\log n}, j_1, \ldots, j_{\log n}\right) = A_{ij}$.

- Let $C = A \cdot B$ denote the true answer.

- Let $\tilde{A}, \tilde{B}$ denote the multilinear extensions of the functions A and B.

MatMult Protocol

- P sends a matrix D claimed to equal $C = A \cdot B$.

MatMult Protocol

- P sends a matrix D claimed to equal $C = A \cdot B$.
- V evaluates $\widetilde{D}$ at a random point $(\boldsymbol{r_1}, \boldsymbol{r_2}) \in \boldsymbol{F}^{\log n} \times \boldsymbol{F}^{\log n}$.

MatMult Protocol

- P sends a matrix D claimed to equal $C = A \cdot B$.
- V evaluates $\widetilde{D}$ at a random point $(\boldsymbol{r_1}, \boldsymbol{r_2}) \in \boldsymbol{F}^{\log n} \times \boldsymbol{F}^{\log n}$.
- By Schwartz-Zippel: it is safe for V to believe that D equals the correct answer C as long as $\widetilde{D}(\boldsymbol{r_1}, \boldsymbol{r_2}) = \tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$.

MatMult Protocol

- P sends a matrix D claimed to equal $C = A \cdot B$.
- V evaluates $\widetilde{D}$ at a random point $(\boldsymbol{r_1}, \boldsymbol{r_2}) \in \boldsymbol{F}^{\log n} \times \boldsymbol{F}^{\log n}$.
- By Schwartz-Zippel: it is safe for V to believe that D equals the correct answer C as long as $\widetilde{D}(\boldsymbol{r_1}, \boldsymbol{r_2}) = \tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$.
- Goal becomes: compute $\tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$

MatMult Protocol

- Goal: Compute $\tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$.

MatMult Protocol

- Goal: Compute $\tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$.
- For Boolean vectors $(\boldsymbol{i}, \boldsymbol{j}) \in \{0,1\}^{\log n}$, clearly:

$$C(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{k} \in \{0,1\}^{\log n}} A(\boldsymbol{i}, \boldsymbol{k}) B(\boldsymbol{k}, \boldsymbol{j})$$

MatMult Protocol

- Goal: Compute $\tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$.
- For Boolean vectors $(\boldsymbol{i}, \boldsymbol{j}) \in \{0,1\}^{\log n}$, clearly:

$$C(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{k} \in \{0,1\}^{\log n}} A(\boldsymbol{i}, \boldsymbol{k}) B(\boldsymbol{k}, \boldsymbol{j})$$

- This implies the following **polynomial** identity:

$$\tilde{C}(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{k} \in \{0,1\}^{\log n}} \tilde{A}(\boldsymbol{i}, \boldsymbol{k}) \tilde{B}(\boldsymbol{k}, \boldsymbol{j}).$$

MatMult Protocol

- Goal: Compute $\tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2})$.
- For Boolean vectors $(\boldsymbol{i}, \boldsymbol{j}) \in \{0,1\}^{\log n}$, clearly:

$$C(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{k} \in \{0,1\}^{\log n}} A(\boldsymbol{i}, \boldsymbol{k}) B(\boldsymbol{k}, \boldsymbol{j})$$

- This implies the following **polynomial** identity:

$$\tilde{C}(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{k} \in \{0,1\}^{\log n}} \tilde{A}(\boldsymbol{i}, \boldsymbol{k}) \tilde{B}(\boldsymbol{k}, \boldsymbol{j}).$$

- So V applies sum-check protocol to compute

$$\tilde{C}(\boldsymbol{r_1}, \boldsymbol{r_2}) = \sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(b_1, \ldots, b_{\log n}),$$

where:

$$g(\boldsymbol{z}) := \tilde{A}(\boldsymbol{r_1}, \boldsymbol{z})\, \tilde{B}(\boldsymbol{z}, \boldsymbol{r_2}).$$

Making V Fast

- At end of sum-check, V must evaluate

$$g(\boldsymbol{r_3}) = \tilde{A}(\boldsymbol{r_1}, \boldsymbol{r_3})\, \tilde{B}(\boldsymbol{r_3}, \boldsymbol{r_2}).$$

- Suffices to evaluate $\tilde{A}(\boldsymbol{r_1}, \boldsymbol{r_3})$ and $\tilde{B}(\boldsymbol{r_3}, \boldsymbol{r_2})$.

Making V Fast

- At end of sum-check, V must evaluate

$$g(\boldsymbol{r_3}) = \tilde{A}(\boldsymbol{r_1}, \boldsymbol{r_3})\, \tilde{B}(\boldsymbol{r_3}, \boldsymbol{r_2}).$$

- Suffices to evaluate $\tilde{A}(\boldsymbol{r_1}, \boldsymbol{r_3})$ and $\tilde{B}(\boldsymbol{r_3}, \boldsymbol{r_2})$.
- Recall that each evaluation can be computed in $O(n^2)$ time.

Making P Fast: A First Attempt

- Recall: we're using sum-check to compute $\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(b_1, \ldots, b_{\log n})$.
- Round i: P sends quadratic polynomial $s_i(X_i)$ claimed to equal: $\sum_{b_{i+1} \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(r_{3,1}, \ldots, r_{3,i-1}, X_i, b_{i+1}, \ldots, b_{\log n})$
  - Suffices for P to specify $s_i(0), s_i(1), s_i(2)$

Making P Fast: A First Attempt

- Recall: we're using sum-check to compute $\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(b_1, \ldots, b_{\log n})$.
- Round i: P sends quadratic polynomial $s_i(X_i)$ claimed to equal: $\sum_{b_{i+1} \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(r_{3,1}, \ldots, r_{3,i-1}, X_i, b_{i+1}, \ldots, b_{\log n})$
  - Suffices for P to specify $s_i(0), s_i(1), s_i(2)$
  - Thus: **Enough for P to evaluate g at all points of the form** $(r_{3,1}, \ldots, r_{3,i-1}, \{0,1,2\}, b_{i+1}, \ldots, b_{\log n})$: $b_{i+1}, \ldots, b_{\log n} \in \{0,1\}^{\log n - i}$

Making P Fast: A First Attempt

- Recall: we're using sum-check to compute $\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(b_1, \ldots, b_{\log n})$.
- Round i: P sends quadratic polynomial $s_i(X_i)$ claimed to equal: $\sum_{b_{i+1} \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(r_{3,1}, \ldots, r_{3,i-1}, X_i, b_{i+1}, \ldots, b_{\log n})$
  - Suffices for P to specify $s_i(0), s_i(1), s_i(2)$
  - Thus: **Enough for P to evaluate g at all points of the form** $(r_{3,1}, \ldots, r_{3,i-1}, \{0,1,2\}, b_{i+1}, \ldots, b_{\log n})$: $b_{i+1}, \ldots, b_{\log n} \in \{0,1\}^{\log n - i}$
  - This is $O(\frac{n}{2^i})$ points.

Making P Fast: A First Attempt

- Recall: we're using sum-check to compute $\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(b_1, \ldots, b_{\log n})$.
- Round i: P sends quadratic polynomial $s_i(X_i)$ claimed to equal: $\sum_{b_{i+1} \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(r_{3,1}, \ldots, r_{3,i-1}, X_i, b_{i+1}, \ldots, b_{\log n})$
  - Suffices for P to specify $s_i(0), s_i(1), s_i(2)$
  - Thus: **Enough for P to evaluate g at all points of the form** $(r_{3,1}, \ldots, r_{3,i-1}, \{0,1,2\}, b_{i+1}, \ldots, b_{\log n})$: $b_{i+1}, \ldots, b_{\log n} \in \{0,1\}^{\log n - i}$
  - This is $O(\frac{n}{2^i})$ points.
  - Recall $\tilde{A}$ and $\tilde{B}$ can each be evaluated at any input in $O(n^2)$ time, and hence so can g.
  - So P can compute s_i in $O\left(\frac{n}{2^i} \cdot n^2\right) = O(n^3/2^i)$ time.

Making P Fast: A First Attempt

- Recall: we're using sum-check to compute $\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(b_1, \ldots, b_{\log n})$.
- Round i: P sends quadratic polynomial $s_i(X_i)$ claimed to equal: $\sum_{b_{i+1} \in \{0,1\}} \cdots \sum_{b_{\log n} \in \{0,1\}} g(r_{3,1}, \ldots, r_{3,i-1}, X_i, b_{i+1}, \ldots, b_{\log n})$
  - Suffices for P to specify $s_i(0), s_i(1), s_i(2)$
  - Thus: **Enough for P to evaluate g at all points of the form** $(r_{3,1}, \ldots, r_{3,i-1}, \{0,1,2\}, b_{i+1}, \ldots, b_{\log n})$: $b_{i+1}, \ldots, b_{\log n} \in \{0,1\}^{\log n - i}$
  - This is $O(\frac{n}{2^i})$ points.
  - Recall $\tilde{A}$ and $\tilde{B}$ can each be evaluated at any input in $O(n^2)$ time, and hence so can g.
- Over **all** rounds, this is $O\left(\sum_i n^3 / 2^i\right) = O(n^3)$ total time.

Making P Fast: Second Attempt

- **Recall: Enough to evaluate g at all points of the form:**
  $\boldsymbol{z} = (r_{3,1}, \dots, r_{3,i-1}, \{0,1,2\}, b_{i+1}, \dots, b_{\log n})$: $b_{i+1}, \dots, b_{\log n} \in \{0,1\}^{\log n - i}$
- Already showed: how to do this in $O\left(n^3/2^i\right)$ time.
- Can we improve this to $O(n^2)$ time?

Making P Fast: Second Attempt

- **Recall: Enough to evaluate g at all points of the form:** $\boldsymbol{z} = (r_{3,1}, \dots, r_{3,i-1}, \{0,1,2\}, b_{i+1}, \dots, b_{\log n})$: $b_{i+1}, \dots, b_{\log n} \in \{0,1\}^{\log n - i}$
- Already showed: how to do this in $O\left(n^3/2^i\right)$ time.
- Can we improve this to $O(n^2)$ time?
- Key observation each entry A_{ij} contributes to $\tilde{A}(\boldsymbol{r_1}, \boldsymbol{z})$ for less than 3 tuples $\boldsymbol{z}$ of the above form.
  - Similarly entry B_{ij} contributes to $\tilde{B}(\boldsymbol{z}, \boldsymbol{r_2})$ for less than 3 tuples $\boldsymbol{z}$ of the above form.

A Fourth Application of the Sum-Check Protocol

A **Better** Doubly-Efficient Interactive Proof for Counting Triangles

A Fourth Application of the Sum-Check Protocol

A **Better** Doubly-Efficient Interactive Proof for Counting Triangles

- In our first IP for counting triangles, V's runtime is $O(n^2)$, P's runtime was $O(n^3)$.
- We now give an IP for the same problem where P runs the fastest known algorithm for counting triangles, and then does $O(n^2)$ extra work to prove the answer is correct.

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ki}$

$$= \frac{1}{6} \cdot \sum_{(i,j) \in [n]^2} (A^2)_{ij} \cdot A_{ij}.$$

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ki}$

$$= \frac{1}{6} \cdot \sum_{(i,j) \in [n]^2} (A^2)_{ij} \cdot A_{ij}.$$

- View A and A^2 as functions mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\boldsymbol{F}$.
- Define the polynomial $h(X,Y) = \widetilde{(A^2)}(X,Y)\, \tilde{A}(X,Y)$.

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ki}$

$$= \frac{1}{6} \cdot \sum_{(i,j) \in [n]^2} (A^2)_{ij} \cdot A_{ij}.$$

- View A and A^2 as functions mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\boldsymbol{F}$.
- Define the polynomial $h(X,Y) = \widetilde{(A^2)}(X,Y)\, \tilde{A}(X,Y)$.
- The Protocol:
  - Apply the sum-check protocol to h.

Counting Triangles

- Input: $A \in \{0,1\}^{n \times n}$, representing the adjacency matrix of a graph.
- Desired Output: $\frac{1}{6} \cdot \sum_{(i,j,k) \in [n]^3} A_{ij} A_{jk} A_{ki}$

$$= \frac{1}{6} \cdot \sum_{(i,j) \in [n]^2} (A^2)_{ij} \cdot A_{ij}.$$

- View A and A^2 as functions mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\boldsymbol{F}$.
- Define the polynomial $h(X,Y) = \widetilde{(A^2)}(X,Y)\, \tilde{A}(X,Y)$.
- The Protocol:
  - Apply the sum-check protocol to h.
  - At the end of the protocol, V needs to evaluate:

$$h(r_1, r_2) = \widetilde{(A^2)}(r_1, r_2)\, \tilde{A}(r_1, r_2).$$

  - V can evaluate $\tilde{A}(r_1, r_2)$ on its own in $O(n^2)$ time. V uses the MatMult protocol to force P to compute $\widetilde{(A^2)}(r_1, r_2)$ for her.